lib/main/nat-less.ath

```
# Ordering properties of natural numbers

load "nat-times"

extend-module N {

declare <: [N N] -> Boolean [[int->nat int->nat]]

module Less {
open Plus

#......................................................................
# Define strict inequality on type N

assert* <-def := [(zero < S n)
                  (~ n < zero)
                  (S n < S m <==> n < m)]

define [zero<S not-zero injective] := <-def

# assert zero<S    := (forall n . zero < S n)
# assert not-zero  := (forall m . ~ (m < zero))
# assert injective := (forall m n . S m < S n <==> m < n)

define [n' x y z] := [?n':N ?x:N ?y:N ?z:N]

#......................................................................

define <S            := (forall n . n < S n)
define =zero         := (forall n . ~ zero < n ==> n = zero)
define zero<         := (forall n . n =/= zero <==> zero < n)
define S1            := (forall x y . S x < y ==> x < y)
define S2            := (forall x y . x < y ==> x < S y)
define S4            := (forall m n . S m < n ==> exists n' . n = S n')
define S-step        := (forall x y . x < S y & x =/= y ==> x < y)
define discrete      := (forall n . ~ exists x . n < x & x < S n)
define transitive    := (forall x y z . x < y & y < z ==> x < z)
define transitive1   := (forall x y z . x < y & ~ z < y ==> x < z)
define transitive2   := (forall x y z . x < y & ~ x < z ==> z < y)
define transitive3   := (forall x y z . ~ y < x & y < z ==> x < z)
define irreflexive   := (forall n . ~ n < n)
define asymmetric    := (forall m n . m < n ==> ~ n < m)
define S-not-<       := (forall n . ~ S n < n)
define Reverse-S     := (forall n m . ~ m < n ==> n < S m)
define trichotomy    := (forall m n . ~ m < n & m =/= n ==> n < m)
define trichotomy1   := (forall m n . ~ m < n & ~ n < m ==> m = n)
define trichotomy2   := (forall m n . m = n <==> ~ m < n & ~ n < m)
define Plus-cancellation := (forall k m n . m + k < n + k ==> m < n)
define Plus-k        := (forall k m n . m < n ==> m + k < n + k)
define Plus-k1       := (forall k m n . m < n ==> m < n + k)
define Plus-k-equiv  := (forall k m n . m < n <==> m + k < n + k)
define not-equal     := (forall m n . m < n ==> m =/= n)
define not-equal1    := (forall m n . m < n ==> n =/= m)

} # Less

#......................................................................

declare <=: [N N] -> Boolean [[int->nat int->nat]]

module Less= {

define [n' x y z] := [?n':N ?x:N ?y:N ?z:N]

assert* <=-def := [(x <= y <==> x < y | x = y)]
#assert <=-def := (forall x y . x <= y <==> x < y | x = y)

define definition := <=-def
```

```
define Implied-by-<     := (forall m n . m < n ==> m <= n)
define Implied-by-equal := (forall m n . m = n ==> m <= n)
define reflexive   := (forall n . n <= n)
define zero<=      := (forall n . zero <= n)
define S-zero-S-n  := (forall n . S zero <= S n)
define injective   := (forall n m . S n <= S m <==> n <= m)
define not-S       := (forall n . ~ S n <= n)
define S-not-equal := (forall k n . S k <= n ==> k =/= n)
define discrete    := (forall m n . m < n ==> S m <= n)
define transitive  := (forall x y z . x <= y & y <= z ==> x <= z)
define transitive1 := (forall x y z . x < y & y <= z ==> x < z)
define transitive2 := (forall x y z . x <= y & y < z ==> x < z)
define S1          := (forall n m . n <= m ==> n < S m)
define S2          := (forall n m . n <= m ==> n <= S m)
define S3          := (forall n . n <= S n)
define trichotomy1 := (forall m n . ~ n <= m ==> m < n)
define trichotomy2 := (forall m n . ~ n < m ==>  m <= n)
define trichotomy3 := (forall m n . n < m ==> ~ m <= n)
define trichotomy4 := (forall m n . n <= m ==> ~ m < n)
define trichotomy5 := (forall m n . m <= n & n <= m ==> m = n)
define Plus-cancellation :=
                      (forall k m n . m + k <= n + k ==> m <= n)
define Plus-k      := (forall k m n . m <= n ==> m + k <= n + k)
define Plus-k1     := (forall k m n . m <= n ==> m <= n + k)
define k-Less=     := (forall k m n . n = m + k ==> m <= n)
define zero2       := (forall n . n <= zero ==> n = zero)
define not-S-zero  := (forall n . ~ S n <= zero)
define S4          := (forall m n . S m <= n ==> exists n' . n = S n')
define S5          := (forall n m . n <= S m & n =/= S m ==> n <= m)
define =zero       := (forall m . m < one ==> m = zero)
define zero<=one   := (forall m . m = zero ==> m <= one)
} # Less=

#.......................................................................
# Proofs

by-induction Less.<S {
  zero => (!chain-> [true ==> (zero < S zero) [Less.zero<S]])
| (S n) => (!chain-> [(n < S n) ==> (S n < S S n) [Less.injective]])
}

datatype-cases Less.=zero {
  zero =>
    assume (~ zero < zero)
      (!reflex zero)
| (S n) =>
    assume A := (~ zero < S n)
      (!from-complements (S n = zero)
        A
        (!chain-> [true ==> (zero < S n) [Less.zero<S]]))
}

conclude Less.zero<
  pick-any n
    (!equiv
     conclude (n =/= zero ==> zero < n)
       (!contra-pos (!instance Less.=zero [n]))
    conclude (zero < n ==> n =/= zero)
     assume (zero < n)
       (!by-contradiction (n =/= zero)
        assume (n = zero)
          (!absurd
           (!chain-> [(zero < n) ==> (zero < zero)  [(n = zero)]])
           (!chain-> [true ==> (~ zero < zero)      [Less.not-zero]])))))

by-induction Less.S1 {
  zero =>
  conclude (forall ?y . S zero < ?y ==> zero < ?y)
    pick-any y
```

```
      assume Less := (S zero < y)
        (!two-cases
          assume (y = zero)
            (!by-contradiction (zero < y)
              assume (~ zero < y)
                let {not-Less :=
                        conclude (~ S zero < y)
                          (!chain->
                           [true ==> (~ S zero < zero) [Less.not-zero]
                                 ==> (~ S zero < y)    [(y = zero)]])}
                   (!absurd Less not-Less))
          assume nonzero := (y =/= zero)
            let {has-predecessor :=
                   (!chain-> [nonzero ==> (exists ?m . y = S ?m)
                                                  [nonzero-S]])}
              pick-witness m for has-predecessor
                (!chain->
                 [true ==> (zero < S m) [Less.zero<S]
                       ==> (zero < y)     [(y = S m)]]))
| (S n) =>
  conclude (forall ?y . S S n < ?y ==> S n < ?y)
    let {induction-hypothesis :=  (forall ?y . S n < ?y ==> n < ?y)}
      pick-any y
        assume Less := (S S n < y)
          (!two-cases
            assume (y = zero)
              (!by-contradiction (S n < y)
                assume (~ S n < y)
                  let {not-Less :=
                         (!chain->
                          [true
                           ==> (~ S S n < zero) [Less.not-zero]
                           ==> (~ S S n < y)    [(y = zero)]])}
                     (!absurd Less not-Less))
           assume nonzero := (y =/= zero)
             let {has-predecessor :=
                    (!chain->
                     [nonzero
                      ==> (exists ?m . y = S ?m) [nonzero-S]])}
               pick-witness m for has-predecessor witnessed
                 (!chain->
                  [(S S n < y)
                   ==> (S S n < S m) [witnessed]
                   ==> (S n < m)     [Less.injective]
                   ==> (n < m)       [induction-hypothesis]
                   ==> (S n < S m)   [Less.injective]
                   ==> (S n < y)     [(y = S m)]]))
}

# It's simpler if we use datatype-cases for the case splitting:

by-induction Less.S1 {
  zero =>
  datatype-cases (forall ?y . S zero < ?y ==> zero < ?y) {
    zero =>
      assume is-Less := (S zero < zero)
        let {is-not-Less := (!chain->
                               [true ==> (~ S zero < zero)
                                            [Less.not-zero]])}
          (!from-complements (zero < zero) is-Less is-not-Less)
  | (S m) =>
       assume (S zero < S m)
         (!chain-> [true ==> (zero < S m) [Less.zero<S]])
  }
| (S n) =>
    let {induction-hypothesis := (forall ?y . S n < ?y ==> n < ?y)}
      datatype-cases (forall ?y . S S n < ?y ==> S n < ?y) {
        zero =>
          assume Less := (S S n < zero)
            let {not-Less := (!chain->
```

```
                             [true ==> (~ S S n < zero)
                                   [Less.not-zero]])}
              (!from-complements (S n < zero) Less not-Less)
      | (S m) =>
          (!chain [(S S n < S m)
                   ==> (S n < m)          [Less.injective]
                   ==> (n < m)            [induction-hypothesis]
                   ==> (S n < S m)        [Less.injective]])
      }
}

datatype-cases Less.S2 {
  zero =>
    conclude (forall ?y . zero < ?y ==> zero < (S ?y))
      pick-any y
        assume (zero < y)
          (!chain-> [true ==> (zero < S y) [Less.zero<S]])
| (S m) =>
    conclude (forall ?y . S m < ?y ==> S m < (S ?y))
      pick-any y
        (!chain [(S m < y)
                 ==> (m < y)       [Less.S1]
                 ==> (S m < S y)   [Less.injective]])
}

by-induction Less.S-step {
  zero =>
    datatype-cases
      (forall ?y . zero < (S ?y) & zero =/= ?y ==> zero < ?y) {
      zero =>
        assume (zero < S zero & zero =/= zero)
         (!from-complements (zero < zero) (!reflex zero) (zero =/= zero))
    | (S y) =>
        assume (zero < (S S y) & zero =/= S y)
          (!chain-> [true ==> (zero < S y) [Less.zero<S]])
    }
| (S x) =>
    let {induction-hypothesis :=
          (forall ?y . x < (S ?y) & x =/= ?y ==> x < ?y)}
      datatype-cases
          (forall ?y . S x < (S ?y) & S x =/= ?y ==> S x < ?y) {
        zero =>
          assume (S x < S zero & S x =/= zero)
            let {Less := (!chain-> [(S x < S zero)
                                    ==> (x < zero)   [Less.injective]]);
                 not-Less := (!chain->
                              [true ==> (~ x < zero)  [Less.not-zero]])}
              (!from-complements (S x < zero) Less not-Less)
      | (S y) =>
          (!chain
           [(S x < (S S y) & S x =/= S y)
            ==> (x < S y & x =/= y) [Less.injective S-injective]
            ==> (x < y)             [induction-hypothesis]
            ==> (S x < S y)         [Less.injective]])
      }
}

by-induction Less.discrete {
  zero =>
    (!by-contradiction (~ exists ?x . zero < ?x & ?x < S zero)
      assume A := (exists ?x . zero < ?x & ?x < S zero)
        pick-witness x for A witnessed
          (!two-cases
            assume (x = zero)
              let {Less := (!chain->
                            [(zero < x) ==> (zero < zero)  [(x = zero)]]);
                   not-Less := (!chain->
                                [true ==> (~ zero < zero) [Less.not-zero]])}
                (!absurd Less not-Less)
            assume (x =/= zero)
```

```
              let {C :=
                     (!chain-> [(x =/= zero) ==> (exists ?m . x = S ?m)
                                [nonzero-S]])}
                pick-witness m for C
                  let {Less := (!chain->
                                [(x < S zero)
                                 ==> (S m < S zero)    [(x = S m)]
                                 ==> (m < zero)        [Less.injective]]);
                       not-Less := (!chain->
                                    [true ==> (~ m < zero)
                                                      [Less.not-zero]])}
                    (!absurd Less not-Less)))
| (S n) =>
    let {induction-hypothesis := (~ exists ?x . n < ?x & ?x < S n)}
       (!by-contradiction (~ exists ?x . S n < ?x & ?x < S S n)
         assume A := (exists ?x . S n < ?x & ?x < S S n)
           pick-witness x for A witnessed
             (!two-cases
               assume (x = zero)
                 let {is-Less := (!chain->
                                  [(S n < x) ==> (S n < zero)
                                   [(x = zero)]]);
                      is-not-Less := (!chain->
                                      [true ==> (~ S n < zero)
                                                   [Less.not-zero]])}
                   (!absurd is-Less is-not-Less)
               assume (x =/= zero)
                 let {C :=
                       (!chain-> [(x =/= zero) ==> (exists ?m . x = S ?m)
                                                      [nonzero-S]])}
                   pick-witness m for C witnessed
                     let {_ := (!chain->
                                [(S n < x)
                                 ==> (S n < S m)    [witnessed]
                                 ==> (n < m)        [Less.injective]]);
                          E := (!chain->
                                [(x < S S n)
                                 ==> (S m < S S n)  [witnessed]
                                 ==> (m < S n)      [Less.injective]
                                 ==> (n < m & m < S n)      [augment]
                                 ==> (exists ?m . n < ?m & ?m < S n)
                                                    [existence]])}
                       (!absurd E induction-hypothesis)))
}

conclude Less.transitive
  let
   {transitive0 :=
          # A version with the easiest-to-induct-on variable first:
            (forall ?z ?x ?y . ?x < ?y & ?y < ?z ==> ?x < ?z);
    _ := by-induction transitive0 {
         zero =>
          pick-any x y
            assume (x < y & y < zero)
              let {not-Less := (!chain->
                                [true ==> (~ y < zero) [Less.not-zero]])}
                (!from-complements (x < zero) (y < zero) not-Less)
         | (S n) =>
            let {induction-hypothesis :=
                   (forall ?x ?y . ?x < ?y & ?y < n ==> ?x < n)}
              pick-any x y
                assume (x < y & y < S n)
                  conclude (x < S n)
                    let {_ := conclude (x < n)
                                (!two-cases
                                   assume (y = n)
                                     (!chain->
                                      [(x < y) ==> (x < n) [(y = n)]])
                                   assume (y =/= n)
                                     (!chain->
```

```
                                      [(y =/= n)
                                       ==> (y < S n & y =/= n) [augment]
                                       ==> (y < n)             [Less.S-step]
                                       ==> (x < y & y < n)     [augment]
                                       ==> (x < n) [induction-hypothesis]]))}
                      (!chain-> [(x < n) ==> (x < S n)  [Less.S2]])
         }}
     pick-any x y z
       (!chain [(x < y & y < z) ==> (x < z) [transitive0]])

by-induction Less.irreflexive {
  zero => (!chain-> [true ==> (~ zero < zero)  [Less.not-zero]])
| (S n) => (!chain-> [(~ n < n) ==> (~ S n < S n)  [Less.injective]])
}

conclude Less.asymmetric
  pick-any x y
    assume (x < y)
      (!by-contradiction (~ y < x)
         assume (y < x)
           let {is-Less := (!chain->
                            [(y < x)
                             ==> (x < y & y < x)     [augment]
                             ==> (x < x)             [Less.transitive]]);
                is-not-Less := (!chain->
                                [true ==> (~ x < x)  [Less.irreflexive]])}
             (!absurd is-Less is-not-Less))

conclude Less.S-not-<
  pick-any n
    (!by-contradiction (~ S n < n)
      assume (S n < n)
        (!absurd
         (!chain-> [(S n < n) ==> (n < n)  [Less.S1]])
         (!chain-> [true ==> (~ n < n)  [Less.irreflexive]])))

by-induction Less.trichotomy {
  zero =>
    pick-any n
      assume (~ zero < n & zero =/= n)
        conclude (n < zero)
          let {has-predecessor :=
                 (!chain->
                  [(zero =/= n)
                   ==> (n =/= zero)       [sym]
                   ==> (exists ?k . n = (S ?k))  [nonzero-S]])}
             pick-witness k for has-predecessor
               let {Less := (!chain->
                             [true ==> (zero < (S k)) [Less.zero<S]
                                   ==> (zero < n)     [(n = (S k))]]);
                    not-Less := (~ zero < n)}
                 (!from-complements (n < zero) Less not-Less)
| (S m) =>
    let {induction-hypothesis :=
          (forall ?n . (~ m < ?n & m =/= ?n) ==> ?n < m)}
      datatype-cases (forall ?n . ~ S m < ?n & S m =/= ?n ==>
                                  ?n < S m)
      { zero =>
          assume (~ S m < zero & S m =/= zero)
            (!chain-> [true ==> (zero < S m) [Less.zero<S]])
      | (S k) =>
          assume A := (~ S m < (S k) & S m =/= (S k))
            (!chain->
             [A ==> (~ m < k & m =/= k) [Less.injective S-injective]
                ==> (k < m)             [induction-hypothesis]
                ==> ((S k) < S m)       [Less.injective]])
      }
}

conclude Less.trichotomy1
```

```
  pick-any m:N n
    assume (~ m < n & ~ n < m)
      (!by-contradiction (m = n)
        (!chain
         [(m =/= n) ==> (~ m < n & m =/= n)  [augment]
                    ==> (n < m)               [Less.trichotomy]
                    ==> (n < m & ~ n < m)     [augment]
                    ==> false                 [prop-taut]]))

conclude Less.trichotomy2
  pick-any m:N n
    let {A := assume (m = n)
                let {C := (!chain->
                            [true ==> (~ m < m)  [Less.irreflexive]])}
                  (!both (!chain-> [C ==> (~ m < n)    [(m = n)]])
                         (!chain-> [C ==> (~ n < m)    [(m = n)]]));
         B := (!chain [(~ m < n & ~ n < m) ==> (m = n)
                                         [Less.trichotomy1]])}
      (!equiv A B)

conclude Less.not-equal
  pick-any m:N n
    assume (m < n)
      (!by-contradiction (m =/= n)
         assume (m = n)
           let {is-not-Less :=
                  (!chain->
                   [(m = n) ==> (~ m < n & ~ n < m) [Less.trichotomy2]
                            ==> (~ m < n)           [left-and]])}
             (!absurd (m < n) is-not-Less))

conclude Less.not-equal1
  pick-any m:N n
    assume (m < n)
      (!by-contradiction (n =/= m)
        assume (n = m)
          let {is-not-Less :=
                 (!chain->
                  [m = n                          [(n = m)]
                     ==> (~ m < n & ~ n < m)    [Less.trichotomy2]
                     ==> (~ (m < n))            [left-and]])}
            (!absurd (m < n) is-not-Less))

conclude Less=.Implied-by-<
  pick-any m n
    (!chain [(m < n) ==> (m < n | m = n)    [alternate]
                     ==> (m <= n)           [Less=.definition]])

conclude Less=.Implied-by-equal
  pick-any m:N n
    (!chain [(m = n) ==> (m < n | m = n)    [alternate]
                     ==> (m <= n)           [Less=.definition]])

conclude Less=.reflexive
  pick-any n
    let {disj := (!right-either (n < n) (!reflex n))}
      (!chain-> [disj ==> (n <= n) [Less=.definition]])

datatype-cases Less=.zero<= {
  zero =>
   (!chain->
    [(zero = zero) ==> (zero <= zero) [Less=.Implied-by-equal]])
| (S n) =>
   (!chain->
    [true ==> (zero < S n)       [Less.zero<S]
          ==> (zero <= S n)      [Less=.Implied-by-<]])
}

datatype-cases Less=.S-zero-S-n {
  zero =>
```

```
    let {disj := (!right-either (S zero < S zero) (!reflex (S zero)))}
      (!chain->
       [disj ==> (S zero <= S zero) [Less=.definition]])
| (S m) =>
    let {Less :=
          (!chain->
           [true ==> (zero < S m)              [Less.zero<S]
                 ==> (S zero < (S S m))        [Less.injective]]);
         disj := (!left-either Less (S zero = (S S m)))}
      (!chain->
       [disj ==> (S zero <= (S S m))   [Less=.definition]])
}

conclude Less=.injective
  pick-any n m
    (!chain
     [(S n <= S m)
      <==> (S n < S m | S n = S m)  [Less=.definition]
      <==> (n < m | n = m)          [Less.injective S-injective]
      <==> (n <= m)                 [Less=.definition]])

conclude Less=.not-S
  pick-any n
    (!by-contradiction (~ S n <= n)
      assume hyp := (S n <= n)
        let {disjunction :=
               (!chain->
                [hyp ==> (S n < n | S n = n) [Less=.definition]])}
         (!cases disjunction
           assume hyp1 := (S n < n)
             let {not-hyp1 := (!chain-> [true ==> (~ hyp1)
                                          [Less.S-not-<]])}
             (!absurd hyp1 not-hyp1)
           assume hyp2 := (S n = n)
             let {not-hyp2 := (!chain->
                               [true ==> (~ hyp2) [S-not-same]])}
             (!absurd hyp2 not-hyp2)))

conclude Less=.S-not-equal
  pick-any k:N n
    assume hyp := (S k <= n)
      let {P := (S n <= n)}
         (!by-contradiction (k =/= n)
           assume (k = n)
             (!absurd
              (!chain-> [hyp ==> P      [(k = n)]])
              (!chain-> [true ==> (~ P) [Less=.not-S]])))

conclude Less=.discrete
  pick-any m n
    assume (m < n)
      (!by-contradiction (S m <= n)
        assume p := (~ S m <= n)
          let {in-between := (exists ?n . m < ?n & ?n < S m)}
            (!absurd
             (!chain->
              [p ==> (~ (S m < n | S m = n))   [Less=.definition]
                 ==> (~ S m < n & S m =/= n)   [dm]
                 ==> (n < S m)                 [Less.trichotomy]
                 ==> (m < n & n < S m)         [augment]
                 ==> in-between                [existence]])
             (!chain->
              [true ==> (~ in-between)   [Less.discrete]]))))

conclude Less=.trichotomy1
  pick-any m n
    (!chain [(~ n <= m)
             ==> (~ (n < m | n = m))    [Less=.definition]
             ==> (~ n < m & ~ (n = m))  [dm]
             ==> (m < n)                [Less.trichotomy]])
```

```
conclude Less=.trichotomy2
  pick-any m n
    (!contra-pos (!instance Less=.trichotomy1 [n m]))

conclude Less=.trichotomy3
  pick-any m:N n
    assume h1 := (n < m)
      (!by-contradiction (~ m <= n)
         assume h2 := (m <= n)
           let {h3 := (!chain->
                        [h2 ==> (m < n | m = n) [Less=.definition]]);
                selfLess :=
                  conclude (n < n)
                    (!cases h3
                      assume h4 := (m < n)
                        (!chain-> [h1 ==> (h1 & h4)     [augment]
                                      ==> (n < n)    [Less.transitive]])
                      assume (m = n)
                        (!chain-> [(n < m) ==> (n < n) [(m = n)]]));
                not-selfLess :=
                  (!chain->
                   [true ==> (~ n < n) [Less.irreflexive]])}
             (!absurd selfLess not-selfLess))

conclude Less=.transitive
  pick-any x:N y:N z
    assume (x <= y & y <= z)
      let {disj1 := (!chain->
                     [(y <= z) ==> (y < z | y = z) [Less=.definition]])}
        conclude (x <= z)
          (!cases disj1
            assume C := (y < z)
              let {disj2 := (!chain->
                             [(x <= y)
                              ==> (x < y | x = y)  [Less=.definition]])}
                conclude (x <= z)
                  (!cases disj2
                    assume (x < y)
                      (!chain->
                       [(x < y) ==> (x < y & C)    [augment]
                                ==> (x < z)        [Less.transitive]
                                ==> (x <= z)       [Less=.Implied-by-<]])
                    assume (x = y)
                      (!chain-> [(y < z)
                                 ==> (x < z)    [(x = y)]
                                 ==> (x <= z)   [Less=.Implied-by-<]]))
            assume (y = z)
              (!chain-> [(x <= y) ==> (x <= z) [(y = z)]]))

conclude Less=.transitive2
  pick-any x:N y:N z:N
    assume (x <= y & y < z)
      conclude (x < z)
        let {D := (!chain->
                   [(x <= y) ==> (x < y | x = y) [Less=.definition]])}
        (!cases D
         assume (x < y)
           (!chain-> [(x < y & y < z) ==> (x < z) [Less.transitive]])
         assume (x = y)
           (!chain-> [(y < z) ==> (x < z) [(x = y)]]))

conclude Less=.transitive1
  pick-any x:N y:N z:N
    assume (x < y & y <= z)
      (!by-contradiction (x < z)
       assume A := (~ x < z)
         (!absurd
          (y <= z)
          conclude (~ y <= z)
```

```
             (!chain->
              [A ==> (z <= x)      [Less=.trichotomy2]
                 ==> (z <= x & x < y)        [augment]
                 ==> (z < y)       [Less=.transitive2]
                 ==> (~ y <= z)    [Less=.trichotomy3]]))))

conclude Less=.S1
  pick-any n:N m
    assume hyp := (n <= m)
      let {disj := (!chain->
                    [hyp ==> (n < m | n = m) [Less=.definition]])}
        conclude (n < S m)
          (!cases disj
            assume p := (n < m)
              (!chain->
               [true ==> (m < S m)           [Less.<S]
                     ==> (p & m < S m)       [augment]
                     ==> (n < S m)           [Less.transitive]])
            assume (n = m)
              (!chain-> [true ==> (m < S m) [Less.<S]
                              ==> (n < S m) [(n = m)]]))

conclude Less=.S2
  pick-any n m
    (!chain [(n <= m) ==> (n < S m)      [Less=.S1]
                      ==> (n <= S m)     [Less=.Implied-by-<]])

conclude Less=.S3
  pick-any n
    (!chain->
     [true ==> (n < S n)        [Less.<S]
           ==> (n <= S n)       [Less=.Implied-by-<]])

conclude Less=.trichotomy4
  pick-any m n
    (!contra-pos (!instance Less=.trichotomy3 [n m]))

conclude Less=.trichotomy5
  pick-any m:N n
    assume (m <= n & n <= m)
      (!by-contradiction (m = n)
        assume (m =/= n)
          let {P1 := (!chain->
                      [(m <= n)
                       ==> (m < n | m = n) [Less=.definition]]);
               P2 := (!chain->
                      [(n <= m)
                       ==> (n < m | n = m) [Less=.definition]])}
            (!cases P1
              assume (m < n)
                (!cases P2
                  (!chain
                   [(n < m)
                    ==> (m < n & n < m)     [augment]
                    ==> (~ n < m & n < m)   [Less.asymmetric]
                    ==> false               [prop-taut]])
                 assume (n = m)
                   (!absurd (!sym (n = m)) (m =/= n)))
              assume (m = n)
                (!absurd (m = n) (m =/= n))))

by-induction Less.Plus-cancellation {
  zero =>
    pick-any m n
      (!chain [(m + zero < n + zero)
               ==> (m < n) [Plus.right-zero]])
| (S k) =>
    let {induction-hypothesis :=
            (forall ?m ?n . ?m + k < ?n + k ==> ?m < ?n)}
    pick-any m n
```

```
      (!chain
       [(m + S k < n + S k)
        ==> (S (m + k) < S (n + k)) [Plus.right-nonzero]
        ==> (m + k < n + k)         [Less.injective]
        ==> (m < n)                 [induction-hypothesis]])
}

conclude Less=.Plus-cancellation
  pick-any k m n
    assume p1 := (m + k <= n + k)
      let {p2 :=
            (!chain-> [p1 ==> (m + k < n + k | m + k = n + k)
                                         [Less=.definition]])}
      conclude (m <= n)
        (!cases p2
          (!chain [(m + k < n + k)
                   ==> (m < n)      [Less.Plus-cancellation]
                   ==> (m <= n)     [Less=.Implied-by-<]])
          (!chain [(m + k = n + k)
                   ==> (m = n)      [Plus.=-cancellation]
                   ==> (m <= n)     [Less=.Implied-by-equal]]))

conclude Less.Plus-k
  pick-any k m n
    assume hyp1 := (m < n)
      let {goal := (m + k < n + k)}
        (!by-contradiction goal
          (!chain
           [(~ goal) ==> (n + k <= m + k) [Less=.trichotomy2]
                     ==> (n <= m)         [Less=.Plus-cancellation]
                     ==> (~ m < n)        [Less=.trichotomy4]
                     ==> (hyp1 & ~ m < n) [augment]
                     ==> false            [prop-taut]]))

# Alternative version, without using taut [K]:

conclude Less.Plus-k
  pick-any k m n
    assume hyp1 := (m < n)
      let {goal := (m + k < n + k)}
        (!by-contradiction goal
          (!chain
           [(~ goal) ==> (n + k <= m + k) [Less=.trichotomy2]
                     ==> (n <= m)         [Less=.Plus-cancellation]
                     ==> (~ m < n)        [Less=.trichotomy4]
                     ==> false     [method (p) (!absurd hyp1 p)]]))

by-induction Less.Plus-k1 {
  zero =>
    pick-any m n
      (!chain [(m < n) ==> (m < n + zero)  [Plus.right-zero]])
| (S k) =>
    let {IH := (forall ?m ?n . ?m < ?n ==> ?m < ?n + k)}
    pick-any m n
      assume (m < n)
        conclude (m < n + (S k))
          (!chain->
           [true ==> (k < (S k))           [Less.<S]
                 ==> (k + n < (S k) + n)   [Less.Plus-k]
                 ==> (n + k < n + (S k))   [Plus.commutative]
                 ==> (m < n & n + k < n + (S k))    [augment]
                 ==> (m < n + k & n + k < n + (S k))     [IH]
                 ==> (m < n + (S k))       [Less.transitive]])
}

conclude Less=.Plus-k
  pick-any k m n
    assume hyp1 := (m <= n)
      let {goal := (m + k <= n + k)}
        (!by-contradiction goal
```

```
          (!chain [(~ goal) ==> (n + k < m + k) [Less=.trichotomy1]
                   ==> (n < m)                  [Less.Plus-cancellation]
                   ==> (~ m <= n)               [Less=.trichotomy3]
                   ==> (hyp1 & ~ m <= n)        [augment]
                   ==> false                    [prop-taut]]))

by-induction Less=.Plus-k1 {
  zero =>
    pick-any m n
      (!chain [(m <= n) ==> (m <= n + zero)  [Plus.right-zero]])
| (S k) =>
    let {IH := (forall ?m ?n . ?m <= ?n ==> ?m <= ?n + k)}
    pick-any m n
      assume (m <= n)
        conclude (m <= n + (S k))
          (!chain->
           [true ==> (k <= (S k))          [Less=.S3]
                 ==> (k + n <= (S k) + n)  [Less=.Plus-k]
                 ==> (n + k <= n + (S k))  [Plus.commutative]
                 ==> (m <= n & n + k <= n + (S k))  [augment]
                 ==> (m <= n + k & n + k <= n + (S k))   [IH]
                 ==> (m <= n + (S k))      [Less=.transitive]])
}

conclude Less.Plus-k-equiv
  pick-any k m n
    (!equiv (!chain [(m < n) ==> (m + k < n + k) [Less.Plus-k]])
            (!chain [(m + k < n + k) ==> (m < n) [Less.Plus-cancellation]]))

by-induction Less=.k-Less= {
  zero =>
    conclude (forall ?m ?n . ?n = ?m + zero ==> ?m <= ?n)
      pick-any m n
        assume hyp := (n = m + zero)
         (!chain->
          [m = (m + zero)     [Plus.right-zero]
             = n              [hyp]
           ==> (m <= n)       [Less=.Implied-by-equal]])
| (S k) =>
   conclude (forall ?m ?n . ?n = ?m + (S k) ==> ?m <= ?n)
     pick-any m n
       let {ind-hyp := (forall ?m ?n . ?n = ?m + k ==> ?m <= ?n)}
         assume hyp :=  (n = m + (S k))
           let {C := (!chain->
                      [n = (m + (S k))    [hyp]
                         = (S (m + k))    [Plus.right-nonzero]
                         = (S m + k)      [Plus.left-nonzero]
                         ==> (S m <= n)   [ind-hyp]])}
            (!chain->
             [true ==> (m <= S m)         [Less=.S3]
                   ==> ((m <= S m) & C)   [augment]
                   ==> (m <= n)           [Less=.transitive]])
}

conclude Less=.zero2
  pick-any n
    assume hyp := (n <= zero)
      let {disj := (!chain->
                     [hyp ==> (n < zero | n = zero) [Less=.definition]])}
      (!cases disj
         assume (n < zero)
           let {not-Less := (!chain->
                              [true ==> (~ n < zero) [Less.not-zero]])}
           (!from-complements (n = zero) (n < zero) not-Less)
         assume (n = zero)
           (!claim (n = zero)))

# Alternative - and much shorter - version using chaining and disjunctive
# syllogism. (This essentially lifts the case analysis from the
# source text and relegates it to the dsyl method.) [K]:
```

```
conclude Less=.zero2
  pick-any n
    assume hyp := (n <= zero)
      (!dsyl (!chain->
              [hyp ==> (n < zero | n = zero) [Less=.definition]])
             (!chain->
              [true ==> (~ n < zero)          [Less.not-zero]]))

conclude Less=.not-S-zero
  pick-any n
    (!by-contradiction (~ S n <= zero)
      assume A := (S n <= zero)
        (!absurd
          (!chain->
           [A ==> (S n = zero)   [Less=.zero2]])
          (!chain->
           [true ==> (S n =/= zero)   [S-not-zero]]))))

conclude Less.Reverse-S
  pick-any n m
    (!chain [(~ m < n)
             ==> (n <= m)     [Less=.trichotomy2]
             ==> (n < S m)    [Less=.S1]])

conclude Less.S4
 conclude  (forall ?m ?n . S ?m < ?n ==> (exists ?n' . ?n = (S ?n')))
  pick-any m n
    assume A := (S m < n)
      let {B := (!chain->
                 [true ==> (n = zero | (exists ?n' . n = (S ?n')))
                        [(!constructor-exhaustiveness "N")]])}
        (!cases B
          assume C := (n = zero)
            (!from-complements (exists ?n' . n = (S ?n'))
             (!chain-> [A ==> (S m < zero) [C]])
             (!chain-> [true ==> (~ S m < zero)
                                 [Less.not-zero]]))
          assume D := (exists ?n' . n = (S ?n'))
            (!claim D))

conclude Less=.S4
 conclude  (forall ?m ?n . S ?m <= ?n ==> (exists ?n' . ?n = S ?n'))
  pick-any m n
    assume A := (S m <= n)
      let {B := (!chain->
                 [A ==> (S m < n | S m = n) [Less=.definition]])}
        (!cases B
         (!chain
          [(S m < n) ==> (exists ?n' . n = (S ?n'))  [Less.S4]])
         (!chain
          [(S m = n) ==> (n = S m)  [sym]
                     ==> (exists ?n' . n = (S ?n')) [existence]]))

conclude Less=.S5
  conclude (forall ?n ?m . ?n <= S ?m & ?n =/= S ?m ==> ?n <= ?m)
  pick-any n m
    let {A1 := (n <= S m);
         A2 := (n =/= S m)}
    assume (A1 & A2)
      let {B := (!chain->
                 [A1 ==> (n < S m | n = S m) [Less=.definition]])}
      (!cases B
        assume B1 := (n < S m)
          (!chain->
           [B1 ==> (S n <= S m)  [Less=.discrete]
               ==> (n <= m)      [Less=.injective]])
        assume B2 := (n = S m)
          (!from-complements (n <= m) B2 A2))
```

```
conclude Less=.=zero
  pick-any m
    (!chain
     [(m < one)
      ==> (~ one <= m)       [Less=.trichotomy3]
      ==> (~ S zero <= m)    [one-definition]
      ==> (~ zero < m)       [Less=.discrete]
      ==> (m = zero)         [Less.=zero]])

conclude Less=.zero<=one
  pick-any m
    assume (m = zero)
      conclude (m <= one)
        (!chain->
         [true ==> (zero <= one) [Less=.zero<=]
               ==> (m <= one)    [m = zero]])

extend-module Times {

define =-cancellation :=
  (forall y z x . zero < x & x * y = x * z ==> y = z)

by-induction =-cancellation {
  zero =>
    pick-any z x
      assume (zero < x & x * zero = x * z)
        conclude (zero = z)
          let {D := (!chain-> [(x * z)
                             = (x * zero)    [(x * zero = x * z)]
                             = zero          [right-zero]
                             ==> (x = zero | z = zero)
                                             [no-zero-divisors]])}
            (!cases D
              assume (x = zero)
                (!from-complements (zero = z)
                   (x = zero)
                   (!chain->
                    [(zero < x) ==> (x =/= zero) [Less.not-equal1]]))
              assume (z = zero)
                (!sym (z = zero)))
| (S y) =>
    let {ind-hyp := (forall ?z ?x . zero < ?x & ?x * y = ?x * ?z ==> y = ?z)}
      datatype-cases (forall ?z ?x .
                       zero < ?x & ?x * (S y) = ?x * ?z ==> (S y) = ?z)
      {
        zero =>
          conclude (forall ?x .
                      zero < ?x & ?x * (S y) = ?x * zero ==> (S y) = zero)
            pick-any x
              assume (zero < x & x * (S y) = x * zero)
                let {C1 := (!chain->
                            [(x * (S y))
                             = (x * zero) [(x * (S y) = x * zero)]
                             = zero       [right-zero]
                             ==> (x = zero | (S y) = zero)
                             [no-zero-divisors]])}
                  (!cases C1
                    assume (x = zero)
                      (!from-complements ((S y) = zero)
                        (x = zero)
                        (!chain->
                         [(zero < x) ==> (x =/= zero) [Less.not-equal1]]))
                    assume ((S y) = zero)
                      (!claim ((S y) = zero)))
      | (S z) =>
          conclude (forall ?x . zero < ?x & ?x * (S y) = ?x * (S z)
                                ==> (S y) = (S z))
            pick-any x
              assume (zero < x & x * (S y) = x * (S z))
                (!chain->
```

```
                  [(x * y + x)
                   = (x * (S y))  [right-nonzero]
                   = (x * (S z))  [(x * (S y) = x * (S z))]
                   = (x * z + x)
                   ==> (x * y = x * z)  [Plus.=-cancellation]
                   ==> (zero < x & x * y = x * z) [augment]
                   ==> (y = z)                    [ind-hyp]
                   ==> (S y = S z)                [S-injective]])
      }
}

define <-cancellation :=
  (forall y z x . zero < x & x * y < x * z ==> y < z)

by-induction <-cancellation {
  zero =>
    pick-any z x
      assume (zero < x & x * zero < x * z)
        (!by-contradiction (zero < z)
          assume A := (~ zero < z)
            let {_ := (!chain-> [A ==> (z = zero) [Less.=zero]])}
              (!absurd
               (!chain->
                [(x * zero < x * z)
                 ==> (zero < zero)  [right-zero (z = zero)]])
               (!chain->
                [true ==> (~ zero < zero) [Less.irreflexive]])))
| (S y) =>
    let {ind-hyp := (forall ?z ?x . zero < ?x & ?x * y < ?x * ?z ==> y < ?z)}
      datatype-cases (forall ?z ?x . zero < ?x & ?x * (S y) < ?x * ?z
                                   ==> (S y) < ?z)
      {
        zero =>
          pick-any x
            assume (zero < x  &  x * (S y) < x * zero)
              (!from-complements ((S y) < zero)
                (!chain-> [(x * (S y) < x * zero)
                           ==> (x * (S y) < zero)  [right-zero]])
                (!chain-> [true ==> (~ x * (S y) < zero) [Less.not-zero]]))
      | (S z) =>
          pick-any x
            assume (zero < x  &  x * (S y) < x * (S z))
              conclude ((S y) < (S z))
                (!chain->
                 [(x * (S y) < x * (S z))
                  ==> (x * y + x < x * z + x) [right-nonzero]
                  ==> (x * y < x * z)         [Less.Plus-cancellation]
                  ==> (y < z)                 [(zero < x) ind-hyp]
                  ==> ((S y) < (S z))         [Less.injective]])
      }
}

define <-cancellation-conv :=
  (forall x y z . zero < x & y < z ==> x * y < x * z)

conclude <-cancellation-conv
  pick-any x y z
    assume A1 := (zero < x & y < z)
      let {goal := (x * y < x * z)}
        (!by-contradiction goal
          assume (~ goal)
            let {D := (!chain->
                       [(~ goal)
                        ==> (x * z <= x * y)    [Less=.trichotomy2]
                        ==> (x * z < x * y | x * z = x * y)
                                                [Less=.definition]])}
              (!cases D
                assume A2 := (x * z < x * y)
                  (!chain->
                   [A2 ==> (z < y)              [<-cancellation]
```

```
                       ==> (~ y < z)           [Less.asymmetric]
                       ==> (y < z & ~ y < z)   [augment]
                       ==> false               [prop-taut]])
                assume A3 := (x * z = x * y)
                  (!absurd
                   (!chain->
                    [(zero < x & A3) ==> (z = y) [=-cancellation]])
                   (!chain-> [(y < z)
                          ==> (~ z = y)   [Less.not-equal1]]))))

define <=-cancellation-conv :=
  (forall x y z . y <= z ==> x * y <= x * z)


conclude <=-cancellation-conv
  pick-any x y z
    assume A := (y <= z)
      let {goal := (x * y <= x * z)}
        (!two-cases
          assume A1 := (zero < x)
            (!by-contradiction goal
              assume (~ goal)
                (!chain->
                 [(~ goal)
                  ==> (x * z < x * y)  [Less=.trichotomy1]
                  ==> (z < y)          [A1 <-cancellation]
                  ==> (~ y <= z)       [Less=.trichotomy4]
                  ==> (A & ~ A)        [augment]
                  ==> false            [prop-taut]]))
          assume A2 := (~ zero < x)
            let {C := (!chain->
                       [true ==> (~ x < zero)      [Less.not-zero]
                             ==> (~ x < zero & A2) [augment]
                             ==> (x = zero)        [Less.trichotomy1]])}
              (!chain<- [goal <== (zero * y <= zero * z)   [C]
                              <== (zero <= zero)  [left-zero]
                              <== true            [Less=.reflexive]]))

define identity-lemma1 :=
  (forall x y . zero < x & x * y = x ==> y = one)
define identity-lemma2 :=
  (forall x y . x * y = one ==> x = one)

conclude identity-lemma1
  pick-any x y
    assume (zero < x & x * y = x)
      (!chain->
       [(x * y = x)
        ==> (x * y = x * one)   [right-one]
        ==> (y = one)           [(zero < x) =-cancellation]])

conclude identity-lemma2
  pick-any x y
    assume A := (x * y = one)
      let {C1 := (!by-contradiction (x =/= zero)
                   assume (x = zero)
                     (!absurd
                      (!chain->
                       [true ==> (zero * y = zero) [left-zero]
                             ==> (x * y = zero)    [(x = zero)]
                             ==> (one = zero)      [A]])
                      (!chain->
                       [true ==> (one =/= zero)    [one-not-zero]])));
           C2 := (!by-contradiction (y =/= zero)
                   assume (y = zero)
                     (!absurd
                      (!chain->
                       [true ==> (x * zero = zero) [right-zero]
                             ==> (x * y = zero)    [(y = zero)]
                             ==> (one = zero)      [A]])
```

```
                      (!chain->
                       [true ==> (one =/= zero)    [one-not-zero]])));
           C3 := (!by-contradiction (~ one < x)
                   assume (one < x)
                     let {_ := (!chain->
                                [C2 ==> (zero < y)  [Less.zero<]])}
                       (!absurd
                        (!chain->
                         [(one < x)
                      ==> (zero < y & one < x)      [augment]
                      ==> (y * one < y * x) [<-cancellation-conv]
                      ==> (one * y < x * y) [commutative]
                      ==> (one * y < one)   [A]
                      ==> (y < (S zero))    [left-one one-definition]
                      ==> (y < (S zero) & y =/= zero) [augment]
                      ==> (y < zero)        [Less.S-step]])
                        (!chain->
                         [true ==> (~ y < zero) [Less.not-zero]])));
           C4 := (!chain->
                  [C3 ==> (~ (S zero) < x)  [one-definition]
                      ==> (x <= (S zero))   [Less=.trichotomy2]
                      ==> (x < (S zero) | x = (S zero))  [Less=.definition]])}
         (!by-contradiction (x = one)
           assume A := (x =/= one)
             (!absurd
              (!chain->
               [A ==> (C4 & A)                [augment]
                  ==> (C4 & x =/= (S zero))   [one-definition]
                  ==> (x < (S zero))          [prop-taut]
                  ==> (x =/= zero & x < (S zero))  [augment]
                  ==> (x < zero)              [Less.S-step]])
              (!chain->
               [true ==> (~ x < zero)        [Less.not-zero]])))

define squeeze :=
  (forall x y . x * y < x ==> y = zero)

conclude squeeze
  pick-any x:N y:N
    assume (x * y < x)
      (!by-contradiction (y = zero)
       assume (y =/= zero)
         let {subgoal := (~ x * y < x);
              B := datatype-cases subgoal on y {
                     zero => (!from-complements (~ x * zero < x)
                              (!reflex zero)
                              (!chain->
                               [(y =/= zero)
                                ==> (zero =/= zero) [(y = zero)]]))
                   | (S y) =>
                       (!chain->
                        [true
                     ==> (~ x * y < zero)     [Less.not-zero]
                     ==> (~ x * y + x < zero + x) [Less.Plus-k-equiv]
                     ==> (~ x * y + x < x)    [Plus.left-zero]
                     ==> (~ x * (S y) < x)    [Times.right-nonzero]])
                   }}
          (!absurd (x * y < x) subgoal))
} # Times

} # N

extend-module N {

define combine-inequalities :=
  method (L)
   match L {
      (list-of (Less= x y) rest) =>
         match (!combine-inequalities rest) {
           (Less= y z) =>
```

```
              (!chain->
                [(x <= y & y <= z) ==> (x <= z) [Less=.transitive]])
        | true => (!claim (Less= x y))
        }
    | [] => (!true-intro)
    }

} # N
```